VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

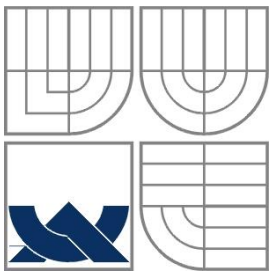
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

HRA V SHADERU

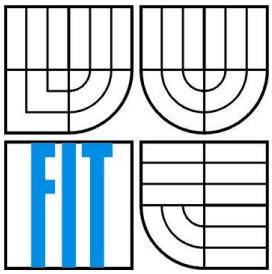
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Ján Ivanecký
AUTHOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

HRA V SHADERU

GAME IN A SHADER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

Ján Ivanecký

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Lukáš Polok

BRNO 2014

Abstrakt

Práca sa zaoberá návrhom a implementáciou jednoduchej strategickej hry pomocou OpenGL, ktorej hernú logiku realizuje grafická karta a zobrazovanie je realizované grafickou kartou v jednom shadery. Práca ďalej obsahuje úvod do OpenGL, prehľad metód tvorby procedurálnych textúr a prehľad zobrazovacích techník založených na metóde Ray Casting.

Abstract

This thesis deals with design and implementation of a simple strategic game using OpenGL, whose game logic is realized by the graphics card and rendering is realized by the graphics card in one shader. It also contains introduction to OpenGL, overview of techniques used for generating procedural textures and overview of rendering methods based on Ray Casting.

Kľúčové slová

OpenGL, shader, procedurálne textúry, herná logika, Perlinov Šum, Voroného diagram, Mandelbrotova množina, Juliova množina, Distance Field Ray Marching, Ray Casting, Celulárny automat

Keywords

OpenGL, shader, procedural textures, game logic, Perlin Noise, Voronoi Diagram, Mandelbrot set, Julia set, Distance Field Ray Marching, Ray Casting, Cellular Automaton

Citace

Ján Ivanecký: Hra v shaderu, bakalářská práce, Brno, FIT VUT v Brně, 2014

Hra v shaderu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením inženýra Lukáša Poloka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ján Ivanecký
21. 5. 2014

Poděkování

Chcel by som poďakovať vedúcemu práce, pánovi Ing. Lukášovi Polokovi, ze jeho usmernenie, pomoc a dohľad nad výsledkom tejto práce.

Obsah

1 Úvod

V počítačových hrách v súčasnosti prevláda rozdelenie práce medzi procesorom a grafickou kartou – procesor sa stará o hernú logiku a grafická karta rieši takmer výlučne zobrazovanie. Herná logika si totiž väčšinou vyžaduje komplexnejšie operácie nad zložitejšími datovými štruktúrami na čo je procesor vhodný, zobrazovanie si na druhej strane vyžaduje veľké množstvo paralelných operácii na čo je stavaná grafická karta. Existujú ale hry, ktorých hernú logiku je možné popísať celulárnym automatom. Celulárne automaty podobne ako zobrazovanie, vyžadujú jednoduchšie operácie prevádzané paralelne vo väčších množstvách a teda je možné takýto automat implementovať na grafickej karte pomocou shaderov.

1.1 Cieľ práce

Cieľom tejto práce je navrhnúť a implementovať pomocou knižnice OpenGL [1] hru, ktorej herná logika sa nepočíta štandardne na procesore, ale je počítaná grafickou kartou. Zobrazenie tejto hry prebieha takisto na grafickej karte, nebude pracovať ale na princípe rasterizácie ako je dnes bežné, ale v metóde Distance Field Ray Marching, ktorá umožňuje celé zobrazovanie v jednom fragment shadery. V ideálnom prípade, by takáto hra mala byť plne programovateľná len pomocou shaderov. Program, v ktorom je vytvorený OpenGL kontext a riadi komunikáciu s grafickou kartou by pri takejto zmene hry nebolo potrebné znovu prekladať.

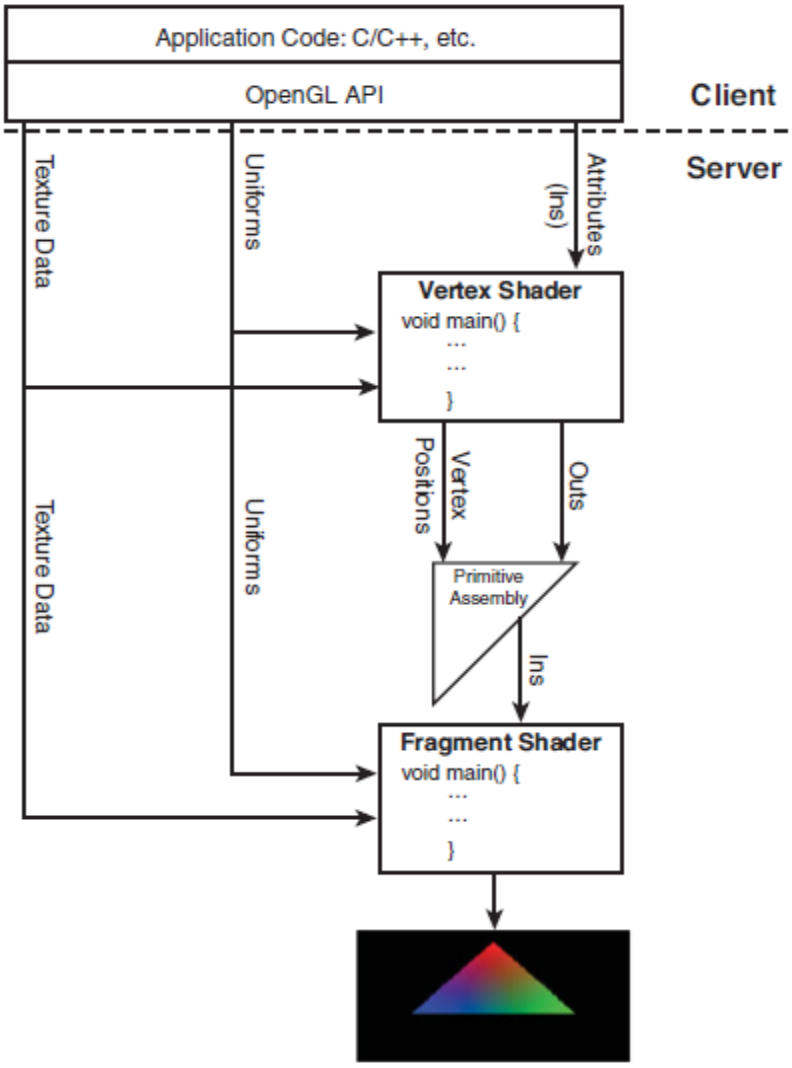
V nasledujúcej kapitole sa nachádza stručný úvod ku knižnici OpenGL. Popisujú sa v nej základné princípy fungovania tejto knižnice a proces zobrazovania používajúci OpenGL rúru. Takisto sa táto kapitola venuje stručnému opisu OpenGL rozšírení a možnostiam vytvárania OpenGL kontextu. Kapitola **3** obsahuje prehľad metód využívaných na procedurálne generovanie textúr, ktoré je možné implementovať v GLSL fragment shadery. Medzi spomínané metódy patrí Perlin Noise, Voroného diagramy, fraktálové metódy založené na Mandelbrotovej a Juliovej množine a Celulárne automaty. Obsahom Kapitoly **4** je návrh fragment shaderu, ktorý realizuje hernú logiku navrhovanej hry. Táto kapitola tiež obsahuje popis hry a spôsobu ukladania stavu hry do textúry. Kapitola **5** sa venuje prehľadu zobrazovacích metód založených na metóde Ray Casting, predovšetkým metóde Distance Field Ray Marching, ktorá je použitá na zobrazovanie implementovanej hry. V závere sa kapitola venuje návrhu fragment shaderu, ktorý bude realizovať zobrazovanie. V kapitole **6** sú popísané implementačné detaily aplikácie ako celku. Ďalej je tu opísaná samotná implementácia fragment shaderu realizújucého hernú logiku. Kapitola sa ďalej venuje implementácii fragment shaderu riešiaceho zobrazovanie a riešeniu problémom ktoré pri

implementácii vznikli. Záver kapitoly porovnáva zobrazovacie metódy Distance Field Ray Marching a Rasterizáciu na implementovanej hre.

2 Rozhranie OpenGL

Knižnica OpenGL [1] (Open Graphics Library) je multiplatformové aplikačné programové rozhranie určené pre akcelerované grafické karty. Rozhranie OpenGL má podobu funkcií v jazyku C, znamená to, že nepodporuje objektový prístup. S datovymi štruktúrami sa nepracuje priamo, sú reprezentované len identifikačnými číslami. OpenGL funguje podobne ako stavový automat, jednotlivými príkazmi sa nastavuje stav celého systému. V nasledujúcom texte je stručne popísaný princíp fungovania OpenGL verzie 3.3.

Základom OpenGL je vykresľovanie primitív – bodov, úsečiek, trojuholníkov do tzv. framebufferu. Spôsob akým sú tieto primitíva vykreslené je špecifikovaný sériou základných príkazov v spolupráci so shadermi. Celý proces vykresľovania jednotlivých primitívov na framebuffer sa nazýva rúra (ang. Pipeline).



Obr. 2.1: *OpenGL pipeline* (prevzaté z [1])

Nutným vstupom takejto pipeline je Vertex Buffer Object (VBO), ktorý obsahuje body v objektovom priestore, ktoré sa majú vykresliť. Na konci pipeline sú tieto body vykreslené na framebuffer.

Významnou časťou tejto pipeline sú práve vertex a fragment shader. Vertex shader určuje spôsob, ktorým budú body z VBO premietané na framebuffer, fragment shader nastavuje farbu a hĺbku jednotlivých bodov framebufferu. Shadre sú programovateľné pomocou jazyka GLSL, ktorý je syntaxou podobný jazyku C. Sú schopné čítať vstupné dáta z riadiaceho programu (v jazyku C/C++) a takisto je možné prenášať dáta z Vertex shaderu do Fragment shaderu.

Framebuffer nemusí byť automatický vykreslený na obrazovku. Jeho obsah je možné zapísať do textúry – takto je možné procedurálne generovať textúry.

2.1 Rozšírenia OpenGL

OpenGL pomocou rozšírení ponúka spôsob rozšírenia funkcionality, ktorú základ OpenGL neobsahuje. Jeden z najjednoduchších spôsobov, akým sa dajú využívať rozšírenia nezávisle od operačného systému je použitie knižnice GLEW (OpenGL Extension Wrangler Library). Tá je schopná určiť, ktoré rozšírenia sú prístupné na cieľovej platforme. Na operačnom systéme Windows taktiež ponúka jednoduchý prístup k funkciám novších verzií OpenGL.

2.2 Vytváranie kontextu

Samotné OpenGL neponúka možnosť vytvorenia kontextu, je nutné vytvoriť ho inak. Na vytvorenie kontextu existuje niekoľko open-source knižníc – SDL, SFML, GLFW, GLUT a iné. Pri implementácii tejto práce je použitá knižnica SDL.

3 Procedurálne generované textúry

Procedurálne generované textúry sú textúry ktoré sú vytvorené nie na základe statických dát, ale na základe matematických vzorcov, alebo algoritmov. Takto generované textúry sa často používajú realistickú reprezentáciu prírodných materiálov ako kameň, drevo, mramor atď. Využívané sú aj napríklad pri tvorbe realistického terénu, kde jednotlivé texely (jeden bod textúry) obsahujú informáciu o výške terénu v danom bode. Veľkou výhodou procedurálnych textúr je prakticky neobmedzené rozlíšenie a takisto malé pamäťové nároky. V prípade nasledujúcej bakalárskej práce budú procedurálne generované textúry využívané ako reprezentácia stavu hry – herná logika. V tejto kapitole budú podrobnejšie popísané súčasne využívané metódy pre tvorbu procedurálnych textúr.

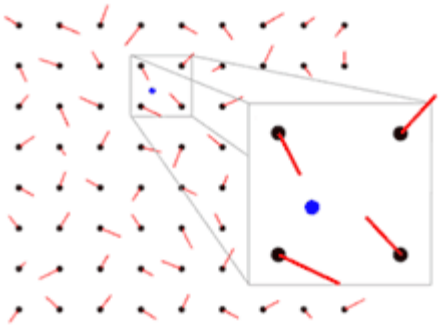
3.1 Perlin noise

Perlin noise [4] je funkcia pre generovanie súvislého pseudonáhodného šumu v priestore. Je používaná hlavne pre zvyšovanie realizmu v počítačovej grafike, keďže imituje kontrolovaný náhodný vzhľad prvkov v prírode – napr. drevo alebo oheň. Pomocou Perlin noise je možné generovať náhodné, realisticky vyzerajúce terény, oblaky, oheň, alebo procedurálne vytvárať textúry pre prírodné materiály ako drevo alebo mramor.

Princíp tejto metódy spočíva vo vytvorení šumovej funkcie pre *n* dimenzií, ktorá je volaná pre každý pixel/jednotku generovaného priestoru. Táto funkcia má *n* reálnych argumentov, a vracia reálne číslo.

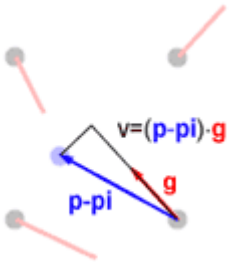
Funkcia pracuje nad vopred vygenerovanou *n* dimenzionálnou mriežkou, ktorá obsahuje na všetkých súradniciach pseudonáhodný gradient s *n* prvkami. Pseudonáhodný znamená, že pre každý celočíselný bod poľa bude gradient rovnaký pri každom výpočte.

Hodnota funkcie v danom bode sa počíta nasledovne: Určíme si do ktorej časti mriežky bod patrí. Na **obr. 3.1** je príklad dvoj-dimenzionálnej mriežky so zaznačenými gradientmi. Modrý bod je bod, pre ktorý chceme zistiť hodnotu šumovej funkcie.



Obr. 3.1: *2D mriežka s pseudonáhodnými gradientmi* (prevzaté z [2])

Pre každý z ohraničujúcich bodov vypočítame skalárny súčin jeho gradientu s vektorom, ktorý určuje vzdialenosť ohraničujúceho bodu od nami preskúmavaného bodu.



Obr. 3.2: *Vektory skalárneho súčinu* (prevzaté z [2])

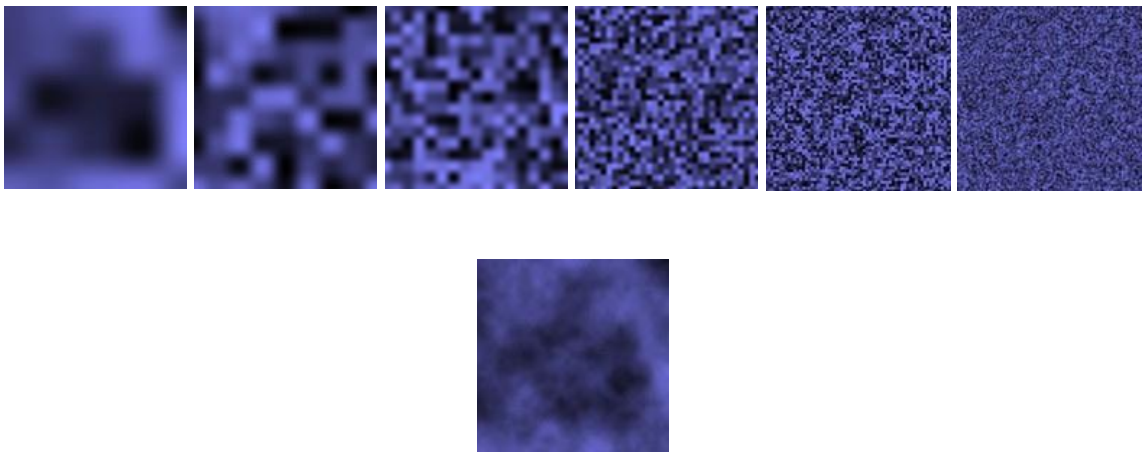
Interpoláciou takýchto skalárnych súčinov dostaneme výslednú hodnotu pre daný bod.

V praxi sa používa skôr prístup, v ktorom sa vygeneruje niekoľko pseudonáhodných šumových funkcií - oktáv, ktoré nemusia byť súvislé a ich spojením vznikne Perlinov šum. Výsledok šumu je závislý na použitých parametroch jednotlivých zložkových funkcií. Parametre sú frekvencia a amplitúda, ktoré sú zvolené pomocou konštanty zvanej perzistencia. Pre jednotlivé funkcie sa frekvencia a amplitúda počíta nasledovne:

$$frekvencia_i = 2^i \tag{3.1}$$

$$amplituda_i = perzistencia^i \tag{3.2}$$

Všeobecne platí, čím viac oktáv, tým detailnejší bude šum. Vzhľadom na obmedzené rozlíšenie textúr existuje hranica počtu oktáv, za ktorou už rozdiel viditeľný nebude.

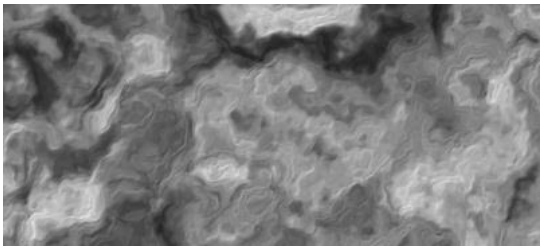


Obr. 3.3: *Šesť oktáv a Perlin Noise z nich zložený* (prevzaté z [3])

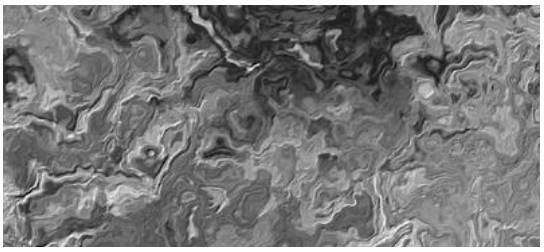
3.1.1 Deformácia definičného oboru

Na základe Perlinovho šumu je možné pomocou deformácie definičného oboru vytvoriť zaujímavé textúry, ktoré sa dajú použiť ako textúry pre kovy, alebo kamene. Perlinov šum sa dá zapísať ako funkcia *p(x)*, kde x je bod, ktorého hodnotu chceme zistiť. Deformácia definičného oboru znamená zmenu bodu x pred jeho vyhodnotením funkciou *p(x)*. Táto deformácia sa dá vyjadriť ako *x + g(x)*, kde *g* je ľubovoľné skreslenie. Funkcia Perlinovho šumu s deformovaným definičným oborom sa dá teda zapísať ako *p(x + g(x))*.

Obr. 3.4: *f = p(x)* (prevzaté z [5])



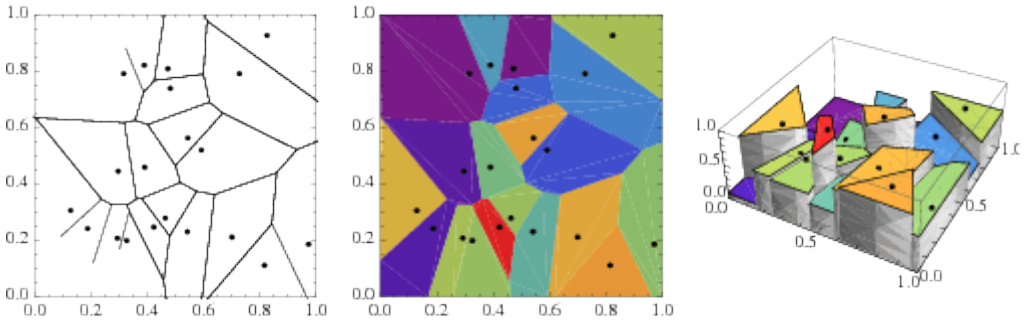
Obr. 3.5: *f = p(x + p(x))* (prevzaté z [5])



Obr. 3.6: *f = p(x + p(x + p(x)))* (prevzaté z [5])

3.2 Voroného diagram

Voroného diagram [6] je metóda rozdelenia priestoru na konvexné oblasti pomocou dopredu definovanou množinou bodov – generátorov. Každá oblasť obsahuje práve jeden takýto generátor a všetky body v tejto oblasti sú bližšie ku tomuto generátoru ako ku všetkým ostatným. Hranice vytvárajú body, ktorých vzdialenosť od dvoch a viacerých generátorov je rovnaká.



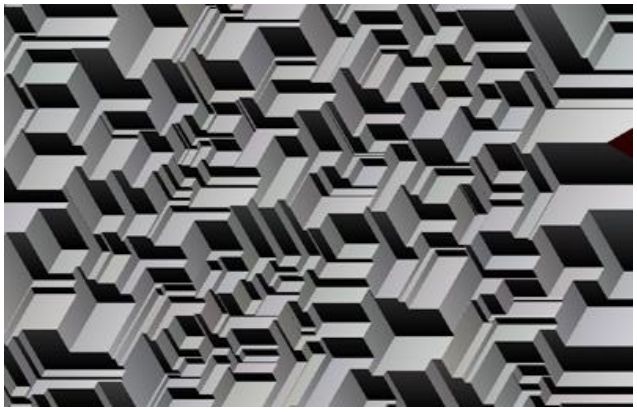
Obr. 3.7: *Voroného diagram v 2D, podľa zafarbenia prevedený do 3D* (prevzaté z [6])

Výsledné rozdelenie na oblasti závisí od použitej metriky pri výpočte vzdialenosti bodov od generátorov. Zrejme najbežnejšie je použitie euklidovskej metriky:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + \ldots + (x_n - y_n)^2}\,, \tag{3.3}$$

Voroného diagramy sú využívané vo veľkom množstve oborov, od matematiky až po epidemiológiu. V počítačovej grafike sa využívajú hlavne na tvorbu procedurálnych textúr pre organické materiály, kamene, alebo lávu.

Obr. 3.8: *Zafarbenie oblastí podľa bitmapy* (prevzaté z [7])



Obr. 3.9: *Voroného diagram s inou ako euklideovskou metrikou* (prevzaté z [7])

3.3 Mandelbrotova a Juliova množina

Mandelbrotova a Juliova množina sú množiny bodov v komplexnej rovine, ktorých hranice tvoria fraktálový tvar. Obe tieto množiny sa určujú vzorkovaním jednotlivých bodov a určením, či ich hodnota smeruje k nekonečnu v prípade iteratívneho použitia jednoduchých matematických operácii. Vzhľadom na to, že takýto iteratívny výpočet sa prevádza pre každý zobrazovaný bod, je vhodné počítať a zobrazovať Mandelbrotovu a Juliovu množinu pomocou shaderov na grafickej karte. Takto vzniknuté procedurálne textúry väčšinou nemajú v praxi iné uplatnenie ako technické demá.

Zobrazenia oboch množín ponúkajú nekonečný detail. Znamená to, že je možné výsledný obraz približovať do nekonečna a množstvo detailu bude stále rovnaké. V praxi ale niečo také nie je možné, keďže výpočtový výkon súčasných počítačov je obmedzený. Za pomoci vylepšení algoritmov už ale existujú videá, kde sú tieto množiny približované niekoľko minút.

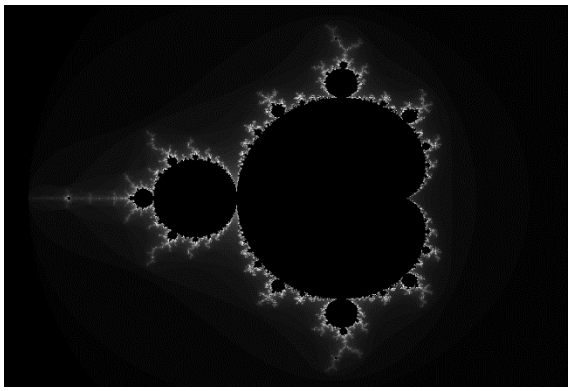
3.3.1 Mandelbrotova množina

Mandelbrotova množina obsahuje body komplexnej roviny, pre ktoré je postupnosť podľa predpisu 3.4 ohraničená, teda platí, že existuje reálne číslo m, také, že pre všetky n je $|z_n| \leq m$.

$$z_{n+1} = z_n^2 + c \tag{3.4}$$

V tomto predpise je *c* skúmaný bod, z_n je výsledok predchádzajúcej iterácie a z_{n+1} je výsledok súčasnej iterácie. Platí, že z_0 je 0 a taktiež, že ak absolútna hodnota z_n prekročí 2, postupnosť ide do nekonečna, teda $m = 2$.

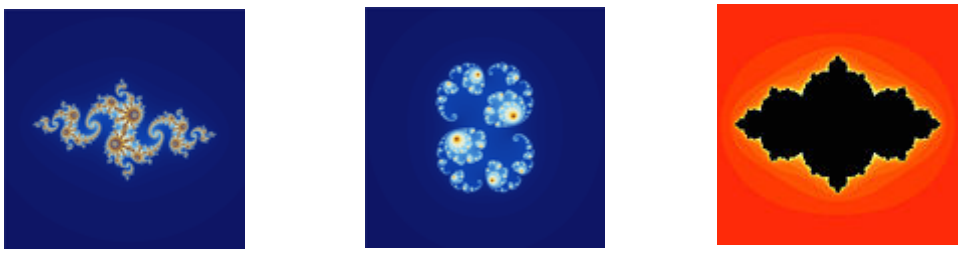
Pri vykresľovaní Mandelbrotovej množiny nepostupuje postupnosť do nekonečna, ale je ukončená po konečnom počte iterácii. Platí, že čím väčší je počet iterácii, zvyšuje sa presnosť zobrazenia a teda aj detail. Pre dosiahnutie efektnejšieho zobrazenia sú zafarbené nie len body pre ktoré je postupnosť ohraničená, ale aj ostatné, na základe rýchlosti s ktorou postupnosť prekročila hodnotu 2.



Obr. 3.10: *Mandelbrotova množina zafarbená monochromaticky*

3.3.2 Juliova množina

Juliova množina je množina bodov komplexnej roviny, pre ktoré je postupnosť podľa predpisu 3.4 ohraničená. Rozdiel oproti Mandelbrotovej množine je, že z_0 nie je 0, ale nami skúmaný bod a *c* je komplexná konštanta, ktorá definuje konkrétnu Juliovu množinu. Spôsob vykresľovania Juliových množín je rovnaký ako pre Mandelbrotovu množinu, čiže detail sa zvyšuje počtom iterácii postupnosti.



c = -0.8 + 0.156i *c = 0.285 + 0.01i* *c = 1 – zlatý rez*

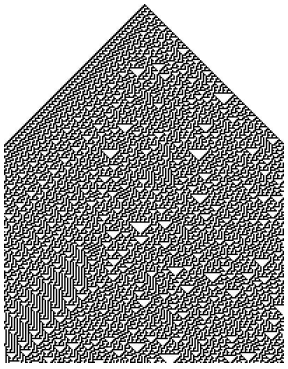
Obr. 3.11: *Zafarbené juliove množiny s rôznymi konštantami c* (prevzaté z [9])

3.4 Celulárne automaty

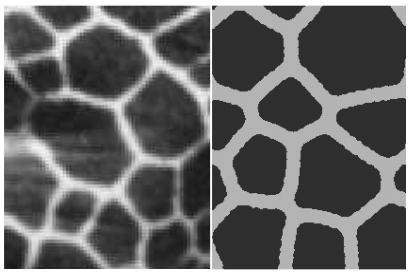
Celulárny automat [10] je priestorový diskrétny model, ktorý pozostáva z N dimenzionálnej sieti buniek, pričom každá z nich môže nadobúdať jeden z konečného počtu stavov. Všetky bunky menia svoj stav v čase podľa pravidla, ktoré nový stav určuje na základe súčasného stavu bunky a stavu jej okolia (okolité bunky).

Celulárne automaty sú veľmi jednoduché na implementáciu a vzhľadom na množstvo paralelných operácii, ktoré sa prevádzajú pri každom kroku v čase, je vhodné implementovať celulárny automat v rámci fragment shaderu, kde každý zobrazovaný pixel predstavuje jednu bunku.

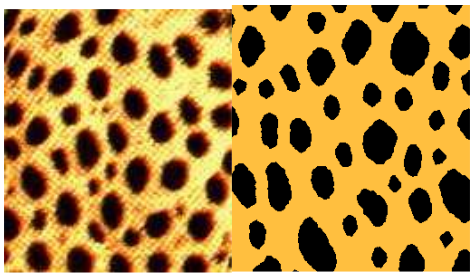
Takto implementované celulárne automaty sa dajú jednoducho použiť na tvorbu textúr, ktoré majú vzor podobný vzoru na koži, alebo ulitách živočíchov.



Obr. 3.12: *Celulárny automat s Pravidlom 30 pripomína ulitu mäkkýšov* (prevzaté z [10])



Obr. 3.13: *Reálna koža žirafy (vľavo), generovaná celul. automatom (vpravo)* (prevzaté z [11])



Obr. 3.14: *Reálna koža geparda (vľavo), generovaná celul. automatom (vpravo)* (prevzaté z [11])

4 Návrh shaderu pre hernú logiku

V tejto kapitole bude popísaný návrh shaderu, ktorý realizuje hernú logiku. Hra je v základe celulárny automat, takže je možné jej hernú logiku implementovať pomocou fragment shaderu a generovať súčasný stav hry ako textúru.

4.1 Popis hry

Hra, ktorá je predmetom bakalárskej práce je ťahová stratégia pre dvoch hráčov. Herné pole je rozdelené na štvorcové políčka, vzniká tým teda mriežka *n x n* buniek. Každý z dvoch hráčov má pridelenú jednu farbu, ktorou sú označené políčka ktoré vlastní a snaží sa obsadiť čo najväčšiu časť hracej plochy políčkami svojej farby pomocou stavby tzv. generátorov. Tie rozširujú svoju farbu na okolité políčka. V prípade konfliktu farieb – jedno políčko je ovplyvňované generátormi oboch hráčov, vyhráva hráč, ktorý viac vplýva na dané políčko – má v jeho blízkosti viacej generátorov alebo jeho generátor je bližšie k políčku ako generátor druhého hráča.

Hráč má možnosť postaviť každý ťah jeden generátor a to na políčku, ktoré nepatrí ani jemu ani súperovi. Cieľom hry je vlastniť čo najväčšie množstvo políčok v momente, keď všetky herné políčka niekomu patria. K tomu je potrebné rozumne rozmiestniť svoje generátory aby pokrývali čo najväčšie územie a mali nad nim čo najväčší vplyv.

V implementácii práce sa celá hra odohráva v 3D priestore, jednotlivé políčka sú zobrazené ako kvádre. Výška kvádra udáva vplyv, ktorý nad týmto políčkom hráč má a jeho farba hráča ktorý ho vlastní.

4.2 Návrh shaderu

Na vytváraní textúry pomocou shaderu sa podieľa prevažne fragment shader, no je potrebný aj vertex shader. Ten slúži na vytvorenie štvorca o veľkosti textúry na celý framebuffer a to pomocou dát vo Vertex Buffer Objekte. Do tohto štvorca sa potom zapisujú farby jednotlivých pixelov, vytvárajúc tak výslednú textúru. Vzhľadom na to, že takáto funkčnosť vertex shaderu je často používaná, nebude tu ďalej popísaná.

Fragment shader realizujúci hernú logiku bude zobrazovať súčasný stav hry do textúry, ktorá bude mať rozlíšenie *n* x *n* (veľkosť hernej plochy). Každý texel takejto textúry teda reprezentuje stav

jedného hracieho políčka a to pomocou ukladania informácii o políčku do jednotlivých farebných zložiek textúry.

Shader dostane ako vstup stavovú textúru, ktorá reprezentuje súčasný stav hry a takisto textúru vstupov, ktorá obsahuje užívateľské vstupy.

Zo stavovej textúry vyčíta shader stav políčka, ktorý je reprezentovaný výškou políčka, hráčom ktorý ho vlastní a jeho typom. Typ udáva, či je na danom políčku postavený generátor, alebo nie. Výška reprezentuje vplyv, ktorý má hráč nad políčkom, pričom maximálnu výšku majú len generátory. Na základe týchto údajov a údajov z textúry vstupu určí shader stav políčka pre ďalší krok.

Textúra vstupov má podobne ako stavová textúra rozlíšenie o veľkosti hracej plochy. Tu jednotlivé texely uchovávajú informáciu, či hráč postavil generátor na príslušnom políčku, a ak áno, ktorý hráč to bol. Na toto stačí jedna farebná zložka textúry. V prípade, že z textúry vstupov vyčíta shader, že niektorý z hráčov postavil na danom políčku generátor, je typ políčka zmenený na generátor a hráč na príslušného hráča.

Určovanie nového stavu políčka je rozdielne pre generátory a bežné políčka. Generátory len zvyšujú svoju výšku až kým nedosiahnu maximálnu hodnotu. Stav políčok iných ako generátory sa určuje podľa hráča a výšky ich susedných políčok vo Von Neumannovskom okolí (priamo susediace políčka, nie diagonálne) a to podľa algoritmu v pseudokóde **Kód 4.1**. Takto vytvorená textúra sa potom v ďalšom kroku používa ako stavová textúra a nový stav hry sa zapisuje do súčasnej stavovej textúry.

```
// Pre súčasné políčko P
Vyska1 = 0; // počítadlo vplyvu pre hráča 1
Vyska2 = 0; // počítadlo vplyvu pre hráča 2

For (susediace políčko N)
{
      If (P.hráč == 1)
            Výška1 += N.výška / K;
      Else
            Výška2 += N.výška / K;
}

If (Výška1 > Výška2)
{
      P.hráč = 1;
      P.výška = Výška1;
} else if (Výška2 > Výška1)
{
      P.hráč = 0;
      P.výška = Výška2;
}
```

Kód 4.1: *Algoritmus pre celulárny automat hernej logiky*

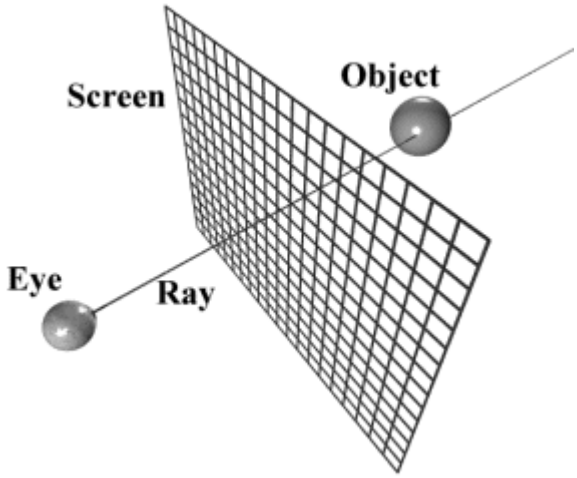
5 Zobrazovanie

V tejto kapitole bude popísaná zobrazovacia metóda Distance Field Ray Marching [14], ktorá bude použitá na zobrazenie hry. Štandardom súčasnosti je vykresľovanie pomocou rasterizácie, keďže ale v tejto práci ide o implementáciu čo najväčšej časti hry v shadery, nie je rasterizácia vyhovujúca, keďže vyžaduje udržiavanie informácie o geometrii sveta na procesore. Pre porovnanie kvality a rýchlosti bude implementované zobrazenie aj pomocou rasterizácie (viď. **6.4**).

5.1 Ray Casting

Ray casting [12] je metóda používajúca detekciu prieniku lúča s povrchom na riešenie rôznych problémov v počítačovej grafike. Je využívaná na zobrazovanie 3D scén, detekciu kolízie objektov, alebo na výpočet osvetlenia jednotlivých plôch scény. Ako zobrazovacia technika dosahuje Ray Casting a metódy na nej založené väčšinou vyššiu kvalitu zobrazenia ako rasterizácia, ale za cenu väčšej výpočtovej náročnosti.

Princíp tejto metódy spočíva vo vysielaní lúčov z kamery, jeden lúč na jeden pixel obrazovky, a sledovaní najbližšieho objektu, ktorý blokuje cestu tomuto lúču. Funguje to teda opačne ako v reálnom svete, kde sa lúče odrážajú od objektu do kamery, resp. oka. Podľa toho, na aký objekt lúč ako prvý narazí sa určí farba príslušného pixelu.

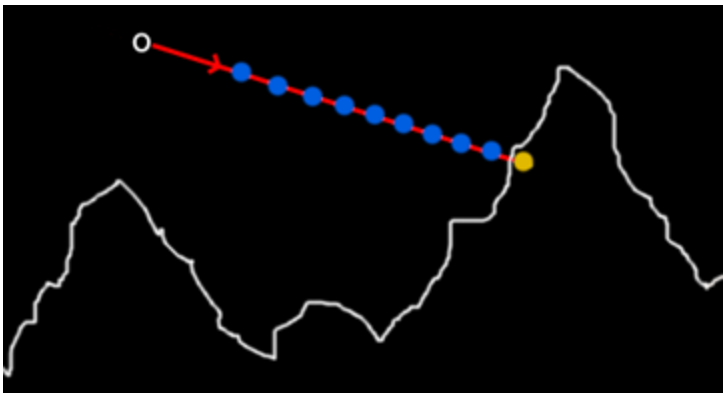


Obr. 5.1: *Princíp metódy Ray Casting – Lúč vedie od oka (eye) cez obrazovku (screen) ku objektu* (prevzaté z [12])

Spôsob detekcie prieniku lúča s objektom je pri klasickom Ray Castingu väčšinou analytický. Každý lúč je popísaný matematickou rovnicou, takisto povrch objektov a je možné týmto spôsobom vypočítať prienik priamky s plochou. Nevýhoda je práve náročnosť výpočtu analytického riešenia rovníc pri väčšom množstve objektov a takisto nutnosť analyticky popísať povrch objektov, čo môže byť v prípade komplexnejších objektov náročné. Riešením takejto situácie je použitie metódy Ray Marching.

5.2 Ray Marching

Ray marching [13] je metóda zobrazovania vychádzajúca z Ray Casting. Princíp vrhania lúčov od kamery cez jednotlivé pixely obrazovky je rovnaký, rozdiel je v detekcii prieniku lúču s povrchom. Kým v Ray Casting je prienik počítaný analyticky, v Ray Marching sa sleduje pohyb lúča po krokoch a po prejdení každého kroku sa kontroluje prienik s povrchom jednotlivých objektov. Platí, že čím menší je krok lúča, metóda je presnejšia, ale výpočtovo náročnejšia. V prípade príliš veľkých krokov môže lúč preskočiť cez objekt alebo povrch a teda sa nezobrazí.



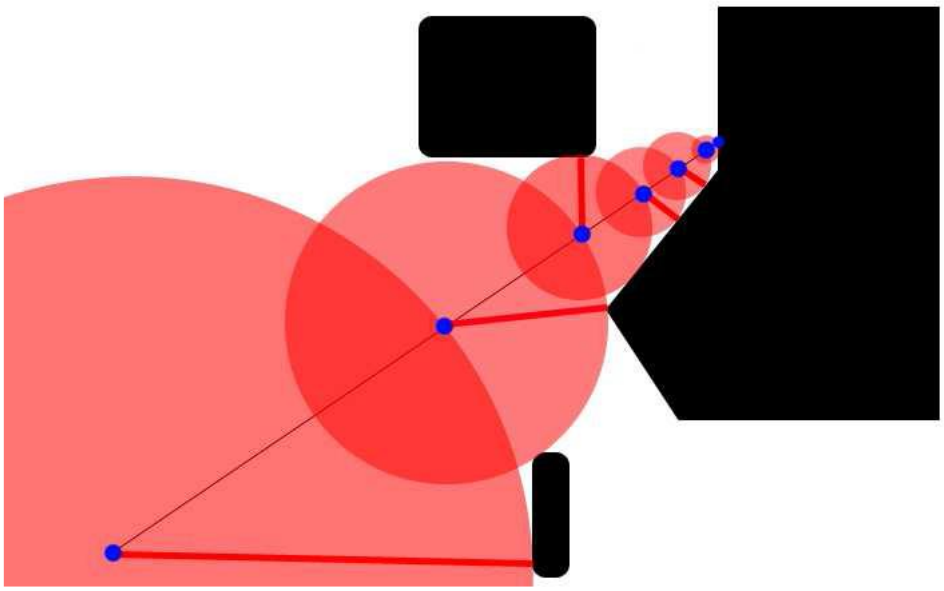
Obr. 5.2: *Krokovanie lúča metódovou Ray Marching* (prevzaté z [13])

Pomocou metódy Ray Marching je možné relatívne jednoducho vykresľovať terény popísané výškovými mapami. Pri krokovaní lúča sa pozerá na pozíciu lúča na mape a potom sa len porovná momentálna výška lúča s výškou terénu v danom bode výškovej mapy.

Vzhľadom na problematickosť určenia veľkosti kroku tak, aby metóda dosahovala požadovanej presnosti s dostatočnou rýchlosťou, boli vyvinuté spôsoby pre adaptívne určenie veľkosti kroku.

5.2.1 Distance Field Ray Marching

Táto metóda vylepšuje Ray Marching o premenlivú veľkosť kroku. Tá je získaná ako vzdialenosť od súčasnej pozície lúča ku najbližšiemu povrchu. Na to je potrebné všetky objekty popísať pomocou analytických funkcií vzdialenosti, ktoré určujú vzdialenosť povrchu objektu k ľubovoľnému bodu virtuálneho sveta. Pomocou tohto určenia veľkosti kroku, sa dosahuje vysoká rýchlosť zobrazenia s vysokou presnosťou, preskakuje sa totiž priestor v ktorom nič nie je.



Obr. 5.3: *Krokovanie lúča metódovou Distance Field Ray Marching* (prevzaté z [14])

Okrem zvýšenia rýchlosti zobrazenia ponúkajú sféry vzdialenosti (distance fields) aj možnosť procedurálne vytvárať virtuálny svet. Nie je potrebné mať informácie o jednotlivých bodoch v priestore, stačí napevno naprogramovať funkcie vzdialenosti pre jednotlivé plochy/objekty. Kombinovaním takýchto funkcií a transformáciou vstupných hodnôt je možné procedurálne vytvárať virtuálny svet.

Pridanou hodnotou tejto metódy je, že ponúka jednoduché prepočítanie svetelných efektov ako sú tiene alebo Ambient Occlusion.

Obr. 5.4: *Funkcia vzdialenosti = styriStlpy(p.x, p.y, p.z)* (prevzaté z [14])

Obr. 5.5: *Funkcia vzdialenosti = styriStlpy(p.x % 1, p.y, p.z % 1)* (prevzaté z [14])

5.3 Návrh shaderu

Vykresľovanie pomocou metódy Ray Marching je podobné s vytváraním textúr. Najprv sa na celú plochu obrazovku vykreslí obdĺžnik, potom sa do neho zapisujú pre každý pixel farby určené algoritmom. Funkčnosť vertex shaderu pre zobrazovanie je rovnaká ako pri shaderi pre hernú logiku.

Samotné zobrazovanie prebieha vo fragment shadery. Ten ako vstup dostane momentálnu pozíciu a rotáciu kamery reprezentovanú maticou, a takisto stavovú textúru vytvorenú shaderom pre hernú logiku. Pre každý pixel vytvorí lúč vychádzajúci od súčasnej pozície kamery, cez príslušný pixel obrazovky. Obrazovka je pri výpočte lúča umiestnená pred kameru, podľa rotácie kamery. Vzdialenosť od obrazovky určuje zorné pole.

Funkcia vzdialenosti v tomto prípade počíta vzdialenosť bodu od každého políčka hracej plochy. Keďže hracie políčko je reprezentované kvádrom, je potrebné počítať vzdialenosť bodu od kvádra n^2 krát a zo všetkých vzdialenosti sa vyberie tá najmenšia. Vzdialenosť bodu v 3D priestore od kvádra, ktorý má stred v bode P(0,0,0), je možné vypočítať pomocou nasledujúcej funkcie:

```
length(max(abs(P) - K, 0.0))
```

Kód. 5.1: *Funkcia vzdialenosti pre kváder v GLSL*

kde K je vektor vzdialenosti stredu kvádra od jeho rohov, funkcia `abs` vráti vektor s absolútnymi hodnotami všetkých zložiek, funkcia `max` vráti vektor, v prípade, že je aspoň jedna jeho zložka kladná, inak vráti 0 a funkcia `length` vráti veľkosť vektoru.

Podľa tohto vzorca je možné zistiť vzdialenosť lúča od jedného herného políčka. Bude sa teda opakovať pre každé herné políčko. Pre každé herné políčko bude potrebné transformovať absolútnu pozíciu lúča na relatívnu pozíciu od stredu políčka, keďže vzorec na výpočet vzdialenosti počíta s kvádrom so stredom v bode P(0,0,0). Na to stačí jednoduché odpočítanie dvoch vektorov.

Po určení vzdialenosti od všetkých herných políčok sa vyberie najmenšia vzdialenosť a lúč spraví krok o veľkosti tejto vzdialenosti. V prípade, že je nejaká vzdialenosť nulová, znamená to, že lúč kolidoval s herným políčkom a určí sa tak farba práve skúmaného pixelu v závislosti od hráča, ktorý políčko vlastní.

Všetky kvádre reprezentujúce herné políčka majú rovnakú šírku a hĺbku, ich výška sa určuje podľa stavovej textúry.

6 Implementácia

V tejto kapitole bude popísaná implementácia navrhnutých shaderov vo forme jednoduchej ťahovej strategickej hry. Hra je implementovaná pomocou knižníc OpenGL a SDL. Z technického hľadiska je zrejme zaujímavejšia implementácia shaderu zobrazenia, a tak jej bude venovaná väčšia časť tejto kapitoly. Prvá verzia hry nepoužívala ako zobrazovaciu metódu Distance Field Ray Marching, ale rasterizáciu, takže časť kapitoly sa bude venovať porovnaniu týchto dvoch zobrazovacích metód.

6.1 Implementácia aplikácie

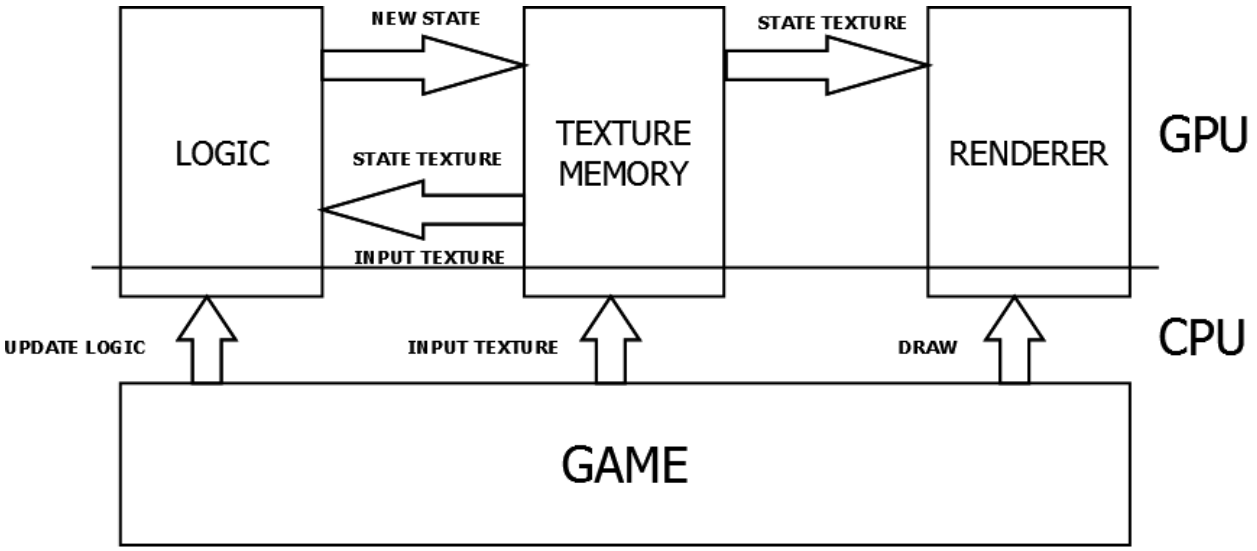
Štandardne hry fungujú na princípe, že logika hry prebieha na procesore a zobrazovanie na grafickej karte. V prípade hry bakalárskej práce herná logika aj zobrazovanie prebieha na grafickej karte. To znamená, že procesor len predáva príslušné informácie grafickej karte a stará sa o zaznamenávanie vstupu.

O priebeh herného cyklu sa teda starajú tri komponenty – `LogicComponent`, ktorý sa stará o hernú logiku, `Renderer`, ktorý sa stará o zobrazovanie a riadiaci komponent `Game`, ktorý obsahuje a riadi predchádzajúce dva.

Hra začína inicializáciou `Game` komponentu, ktorá obsahuje inicializáciu komponentov `LogicComponent` a `Renderer`. Nasleduje samotný beh aplikácie, ktorý pozostáva zo spracovania vstupu a striedavého volania funkcií `LogicComponent` a `Renderer`. Spracovanie vstupu prebieha v komponente `Game` na procesore, keďže grafická karta podobné možnosti nemá. Užívateľský vstup ovplyvňuje stav hry alebo hernú kameru, takže informácie o vstupe sú zapísané do textúry vstupu a tzv. *View* matice. Textúra vstupu obsahuje informácie o tom ktoré políčko bolo stlačené ktorým hráčom a *View* matica obsahuje informácie o súčasnom pohľade – pozícií a rotácii kamery v scéne. Vstupná textúra je potom predávaná do `LogicComponent`-u, *View* matica do `Renderer`-u.

V `LogicComponent` prebehne prvý prechod cez OpenGL pipeline (rúru), kde sa vykreslí herný stav do textúry. V tomto komponente sú pripravené dve textúry s framebuffermi, ktoré sa pomocou Ping-Pong metódy striedajú ako vstupné a výstupné textúry. Každý párny priechod je ako vstupná textúra použitá textúra A a výstup sa zapisuje do textúry B. Pri nepárnych prechodoch je to presne naopak. Ďalším vstupom pre `LogicComponent` je vyššie spomínaná textúra vstupu. Tá ma rozmery rovnaké ako stavová textúra a vo fragment shadery sa zisťujú hodnoty texelu vstupnej

textúry pre rovnaký texel stavovej textúry. Výstupom `LogicComponent` je stavová textúra, ktorá je okamžite predaná komponente `Renderer` cez komponentu `Game`.



Obr. 6.1: *Komunikácia jednotlivých komponentov a ich umiestnenie na procesore a grafickej karte*

V `Renderer` prebieha druhý prechod cez OpenGL pipeline, v ktorom sa vykresľuje samotná scéna. Vstupom pre `Renderer` je *View* matica a stavová textúra hry, podľa ktorých sa vo fragment shadery prevedie vykreslenie scény.

6.1.1 Spracovanie vstupu

Hráčov vstup do hry pozostáva len zo stavania generátorov. Kontrola správnosti vstupu pozostáva z kontroly, či hráč stavia generátor na povolený typ políčka a či stavia vo svojom ťahu. To prebieha na procesore okamžite pri zaregistrovaní vstupu. Ak nie je podmienka pre správny vstup splnená, vstup je odignorovaný a fragment shader sa ani nedozvie, že nastal. Tento prístup znamená oddelenie istej časti hernej logiky od grafickej karty do procesora, ale vzhľadom na to, že je táto časť hernej logiky úzko spätá so spracovaním vstupu, bola by implementácia na grafickej karte nepraktická a komplikovaná.

6.2 Herná logika

Herná logika je implementovaná podľa návrhu z bodu **4.2**, znamená to, že fragment shader pracuje ako celulárny automat.

Samotný proces vyhodnotenia nového stavu pre herné políčko pozostáva zo štyroch krokov. Najskôr sa načíta súčasný stav políčka zo stavovej textúry. Potom prebieha vyhodnotenie nového stavu pre políčko na základe susedných políčok a súčasného stavu. Tretí krok pozostáva z kontroly textúry vstupu – či hráč nepostavil na danom poličku generátor. Na záver sa zapíše nový stav políčka do textúry.

Veľkosť hracej plochy v implementácii je 64x64, znamená to, že stavová textúra a takisto aj textúra vstupu bude mať tento rozmer.

6.2.1 Načítanie stavu z textúry

Stav políčka je reprezentovaný tromi parametrami – výška, hráč a typ. Výška políčka hovorí o vplyve hráča na dané políčko. O ktorého hráča sa jedná samozrejme určuje parameter hráč. Typ políčka určuje či sa jedná o generátor, alebo o štandardné políčko. Textúra je vhodná na uloženie takýchto troch parametrov, keďže bežne obsahuje tri až štyri farebné zložky (RGB, alebo RGBA). Stavová textúra má teda OpenGL formát `RGB`. Pri načítaní stavu pre políčko sú jednotlivé parametre uložené do float premenných.

```
float x = gl_FragCoord.x / mapsize;
float y = gl_FragCoord.y / mapsize;
vec2 position = vec2(x,y);
float height = texture2D(logicTex, position).x;
float player = texture2D(logicTex, position).y;
float type = texture2D(logicTex, position).z;
```

Kód 6.1: *Ukážka spracovania parametrov z textúry v GLSL*

Výška je číslo s hodnotou v intervale 0 až 1. Výška 0 znamená žiadny vplyv od oboch hráčov, výška 1 je najväčšia možná, dosahujú ju prakticky len generátory. Parameter hráč nadobúda len tri hodnoty – 0, 0.5 a 1. 1 reprezentuje hráča č.1, 0.5 hráča č.2 a 0 žiadneho t.j. políčko je prázdne. Typ políčka nadobúda len hodnoty 0 a 1. 0 znamená bežné políčko, 1 znamená generátor.

6.2.2 Vyhodnotenie nového stavu

Vyhodnotenie nového stavu prebieha v závislosti od typu políčka. V prípade, že sa jedná o generátor, mení sa len výška políčka a to tak, že sa zväčšuje ak je menšia ako 1.

Nový stav pre ostatné políčka sa vyhodnocuje na základe algoritmu ktorý je popísaný v **Kóde 4.1**. Konštanta `K` z **Kódu 4.1** je vo výslednom kóde stanovená na 4.

6.2.3 Kontrola vstupu

V prípade, že políčko nie je generátor, kontroluje sa, či ho hráč nepostavil na danej pozícii. Ak áno, nastavia sa parametre typ a hráč. Parameter typ sa nastaví na hodnotu 1, čo značí generátor a hráč sa nastaví podľa hodnoty v textúre vstupu. Textúra vstupu potrebuje uchovať pre každé políčko jednu informáciu - ktorý hráč generátor postavil. Formát textúry vstupu je teda `GL_RED`. Kontrola, či hráč stavia generátor na vhodný typ políčka prebieha na procesore, pri spracovaní vstupu (viď **6.1**).

6.2.4 Zápis nového stavu

Parametre nového stavu sa zapíšu do textúry ako jednotlivé zložky farieb v rovnakom formáte ako pri načítavaní pôvodného stavu.

6.3 Zobrazovanie

Zobrazovanie prebieha metódou Distance Field Ray Marching, teda proces zobrazovania začína vykreslením obdĺžnika cez celú obrazovku a následným vyplnením jednotlivých pixelov vo fragment shadery. Vstupom pre fragment shader je stavová textúra, a *View* matica ktorá popisuje momentálne natočenie a posuv scény.

6.3.1 Inicializácia scény

Proces zobrazovania začína „umiestnením“ oka a zobrazovacej plochy do scény a nastavením východzej pozície a smeru lúča pre daný pixel (pre lepšie pochopenie viď **obr.5.1**). Zobrazovacia plocha je v scéne posunutá v ose y a z, oko je potom posunuté od plochy v ose z.

```
vec3 screen = vec3(0,0,0);
vec3 eye = vec3(0, 0, 2) + screen;
```

Kód 6.2: *Inicializácia pozície oka a zobrazovacej plochy v GLSL*

Ďalej je potrebné určiť počiatok a smer lúča. Počiatok je určený ako bod na zobrazovacej ploche v scéne, ktorý odpovedá pixelu reálnej zobrazovacej plochy. Smer sa určuje na základe rozdielu pozície pixelu na zobrazovacej ploche (počiatok) a oka. Na to, aby bolo možné ovládať pozíciu a smer pohľadu je potrebné vynásobiť vektory reprezentujúce pozíciu oka a počiatok lúča *View* maticou,

ktorá bola do fragment shaderu zaslaná z procesoru a je menená na základe uživateľského vstupu. Normalizovaný rozdiel počiatku lúča a oka určuje smer lúča.

```
vec3 up = vec3(0, 1, 0);
vec3 right = vec3(1, 0, 0);
float x = 1024.0/600.0;
float u = (gl_FragCoord.x * 2.0 / 1024 - 1.0) * x;
float v = gl_FragCoord.y * 2.0 / 600 - 1.0;
vec3 ro = right * u + up * v + screen;
ro = (View * vec4(ro.xyz, 1.0f)).xyz;
eye = (View * vec4(eye.xyz, 1.0f)).xyz;
vec3 rd = normalize(ro - eye);
```

Kód 6.3: *Postup výpočtu počiatku a smeru lúča v GLSL*

6.3.2 Ray Marching

Nasleduje samotný proces krokovania lúča vo vypočítanom smere. Veľkosť kroku sa určuje na základe vzdialenosti k najbližšiemu povrchu scény. V rámci optimalizácie je vhodné si určiť obmedzenia, ktoré zabránia zbytočne zdĺhavému krokovaniu. V tejto implementácii sú tieto obmedzenia maximálny počet krokov, ktoré môže lúč spraviť a takisto maximálna vzdialenosť ktorú môže lúč prejsť.

To, či lúč na niečo narazil je dané funkciou vzdialenosti, ktorá popisuje vzdialenosť bodu v priestore od povrchu zobrazovaného objektu. V prípade, že je vzdialenosť od povrchu menšia ako určitá hranica, dá sa predpokladať, že lúč na povrch narazil. V tom prípade bude pixel zafarbený bielou farbou. Ako príklad bude funkcia vzdialenosti funkcia popisujúca povrch gule s polomerom 1.

```
float surface(vec3 p)
{
      return length(p) – 1.0f;
}
```

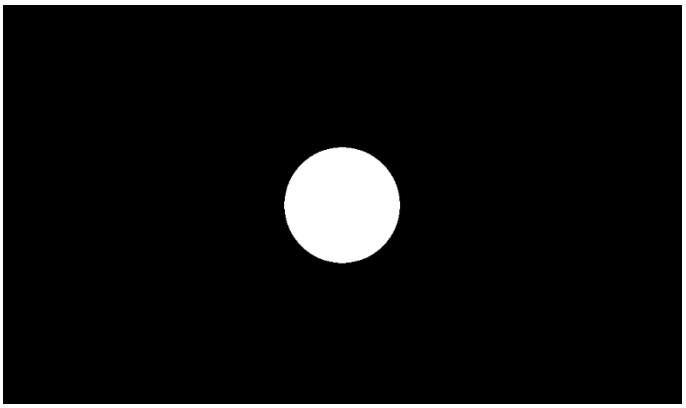
Kód 6.4: *Funkcia vzdialenosti pre guľu s polomerom 1*

Proces Ray Marching-u je popísaný nasledujúcim kusom GLSL kódu:

```
const int maxSteps = 180;
const int maxDist = 150;
float tt = 0.0;               // počítadlo vzdialenosti
float d = 0.0;                // vzdialenosť od povrchu
color = vec4(0.0f);
for(int i = 0; i < maxSteps; ++i)
{
      vec3 p = eye + rd * tt; // krok lúča
      vec3 d = surface(p);
      if(d < 0.001)           // lúč narazil
      {
            color = vec4(1.0f);
            break;
      }
      if(tt > maxDist)        // lúč prešiel maximálnu vzdialenosť
      {
            break;
      }
      tt += d;
}
```

Kód 6.5: *Distance field ray marching v GLSL*

Výsledok vyzerá nasledovne:



Obr. 6.2: *Guľa vykreslená metódou Distance Field Ray Marching*

6.3.3 Lambertov osvetľovací model

V rámci aplikácie je použitý ako osvetľovací model Lambertov osvetľovací model. Je preto potrebné v každom bode vypočítať normálu povrchu. Gradient funkcie vzdialenosti určuje smer najväčšieho rastu v danom bode a to je v podstate normála ktorú chceme dostať. To je možné pomocou Numerickej derivácie za použitia vzorca:

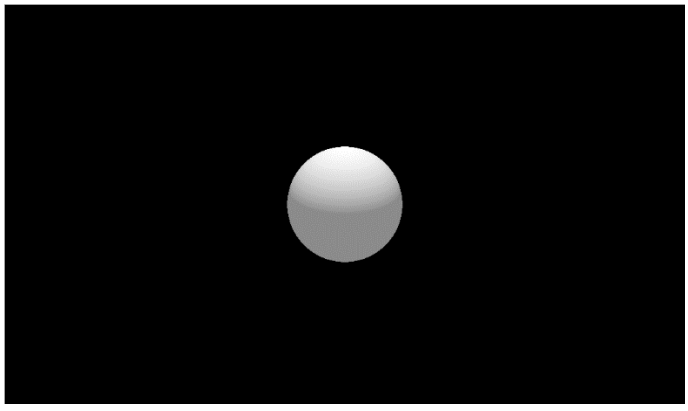
$$f' \cong \frac{f(x+h) - f(x)}{h} \tag{6.1}$$

Vzhľadom na to, že potrebujeme iba smer rastu, nie rýchlosť, nie je potrebné delenie hodnotou h, stačí výsledný vektor znormalizovať. V implementácii práce vyzerá aproximácia normály nasledovne:

```
float gradX = surface(p + vec3(0.01, 0, 0)) - surface(p);
float gradY = surface(p + vec3(0, 0.01, 0)) - surface(p);
float gradZ = surface(p + vec3(0, 0, 0.01)) - surface(p);
vec3 normal = vec3(gradX, gradY, gradZ);
normal = normalize(normal);
```

Kód. 6.6: *Aproximácia normály funkcie vzdialenosti v GLSL*

Po implementácii Lambertovho osvetľovacieho modelu vyzerá výstup nasledovne:



Obr. 6.3: *Lambertov model*

6.3.4 Funkcia vzdialenosti pre scénu

Herná scéna sa skladá z hracej plochy a zeme. Takisto funkcia vzdialenosti pre scénu sa skladá z funkcie vzdialenosti pre hraciu plochu a funkcie vzdialenosti pre zem. Pri vyhodnotení funkcie vzdialenosti sa berie menšia hodnota z hodnôt funkcie vzdialenosti pre hraciu plochu a funkcie vzdialenosti pre zem.

6.3.5 Funkcia vzdialenosti pre hraciu plochu

Funkcia vzdialenosti pre hraciu plochu pozostáva z vhodných transformácii súčasnej polohy lúča tak, aby sa za použitia funkcie vzdialenosti pre jedno herné políčko, vykreslilo herných políčok viac. V základe stačí na opakované zobrazenie objektu v intervale *I* orezať pozíciu lúča v danom smere pomocou funkcie modul do intervalu (–x / 2, x / 2). Výsledkom je nekonečne opakujúci sa objekt.

Pri vykresľovaní hracej plochy ale nie je požiadavka opakovať objekt do nekonečna, ale obmedzený krát. Takisto vykresľované objekty nebudú rovnakých rozmerov.

Podstava kvádra hracieho políčka má rozmery 2x2, výška je premenlivá. To znamená, že v prípade zobrazenia iba jedného políčka, bude v *x*-ovej a *z*-ovej osi ležať na intervale (-1,1). V prípade zobrazenia všetkých políčok (rozmery hracej plochy v implementácii sú 64x64), budú ležať na intervale (-1, 127). Keďže bez akýchkoľvek úprav sa zobrazí kváder na intervale (-1, 1), netreba v tomto intervale nič meniť. V GLSL vyzerá implementácia nasledovne:

```
if (p.x < 127 && p.x > 1)
      p.x = mod(p.x - 1, 2) - 1.0f;
if (p.z < 127 && p.z > 1)
      p.z = mod(p.z - 1, 2) - 1.0f
```

Kód. 6.7: *Opakovanie herného políčka v intervale (1, 127) v x-ovej a z-ovej ose*

Problém pri takejto implementácii nastane v prípade, že sa pozeráme na scénu v zápornom smere *z*-ovej, alebo *x*-ovej osi. V tom prípade lúče ktoré vychádzajú z obrazovky majú pozíciu na danej osi väčšiu ako 127 a teda sa na nich pri kontrole vzdialenosti od povrchu nevzťahuje transformácia pozície. To znamená, že považujú za najbližší povrch práve kváder zo stredom v bode (0,0,0). Preto sa stane, že lúč preskočí jemu najbližšie herné políčko a nezobrazí ho. To rieši modifikácia, ktorá všetky lúče s pozíciou väčšou ako 125 posunie o -126 v danej osi. To znamená, že pre tieto lúče je napevno vytvorené herné políčko s najväčšou pozíciou v danej osi, ktoré nebudú ignorovať.

```
if (p.x < 125 && p.x > 1)
      p.x = mod(p.x - 1, 2) - 1.0f;
if (p.x > 125)
      p.x -= 126.0f;
if (p.z < 125 && p.z > 1)
      p.z = mod(p.z - 1, 2) - 1.0f;
if (p.z > 125)
      p.z -= 126.0f;
```

Kód. 6.8: *Modifikované opakovanie herného políčka v intervale (1, 127) v x-ovej a z-ovej osi*

Po vhodnej transformácii pozície lúča sa vypočítava vzdialenosť od povrchu kvádra so stredom v bode (0,0,0). Keďže ale nie sú všetky herné políčka rovnakej výšky, je potrebné výšku najbližšieho herného políčka vyčítať zo stavovej textúry:

```
float height = texture2D(state, vec2(round(p.x/2) / 64,round(p.z/2) / 64)).x;
```

Kód. 6.9: *Načítanie výšky hracieho políčka zo stavovej textúry*

6.3.6 Funkcia vzdialenosti pre hracie políčko

Hracie políčko je kváder, funkcia vzdialenosti pre kváder je **Kód 5.1**. Táto funkcia ráta s kvádrom, ktorý ma stred v bode (0,0,0). To znamená, že pri zmene výšky kvádra sa zmení jeho veľkosť v kladnom aj zápornom smere osi *y*. Hracie políčko sa ale zväčšuje len v kladnom smere, preto je potreba posunúť kváder v ose y o polovicu jeho celkovej výšky. Tak bude jeho spodná hrana vždy na úrovni bodu 0 osi *y*. V GLSL vyzerá funkcia vzdialenosti takto (vektor b obsahuje vzdialenosť stredu kvádra od jeho rohov):

```
float box( vec3 p, vec3 b )
{
 p.y -= b.y;
 return length(max(abs(p)-b,0.0));
}
```

Kód. 6.10: *Funkcia vzdialenosti pre herné políčko*

6.3.7 Čítanie farieb herných políčok zo stavovej textúry

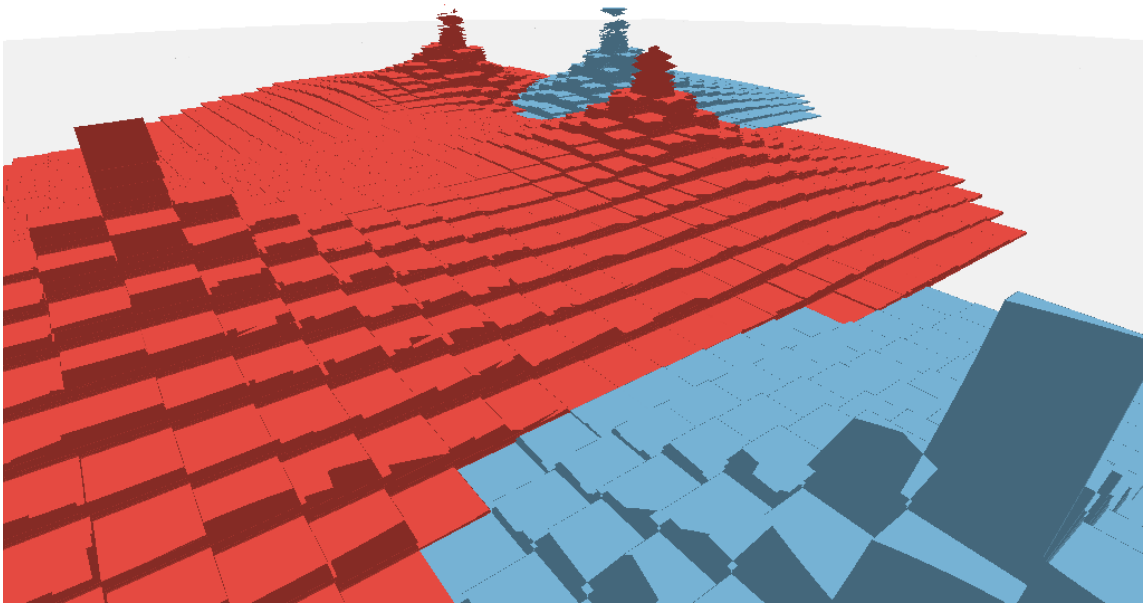
V prípade prieniku lúča s povrchom niektorého z herných políčok, je potrebné zistiť ktorý hráč ho vlastní. Na základe toho, sa určí jeho farba. Načítanie hráča vlastniaceho dané políčko zo stavovej textúry je možné integrovať priamo do funkcie vzdialenosti hracej plochy a teda bude vracať dvojrozmerný vektor, ktorý obsahuje informáciu o vzdialenosti k najbližšiemu povrchu a takisto o farbe tohoto povrchu.

6.3.8 Funkcia vzdialenosti pre zem

Funkcia vzdialenosti pre zem je funkcia vzdialenosti pre rovinu. Pre rovinu, ktorá je kolmá na os *y* a pretína ju v bode 0, vyzerá funkcia vzdialenosti nasledovne:

```
float ground(vec3 p)
{
	return p.y;
}
```

Kód. 6.11: *Funkcia vzdialenosti pre zem*



Obr. 6.4: *Výsledná scéna hry*

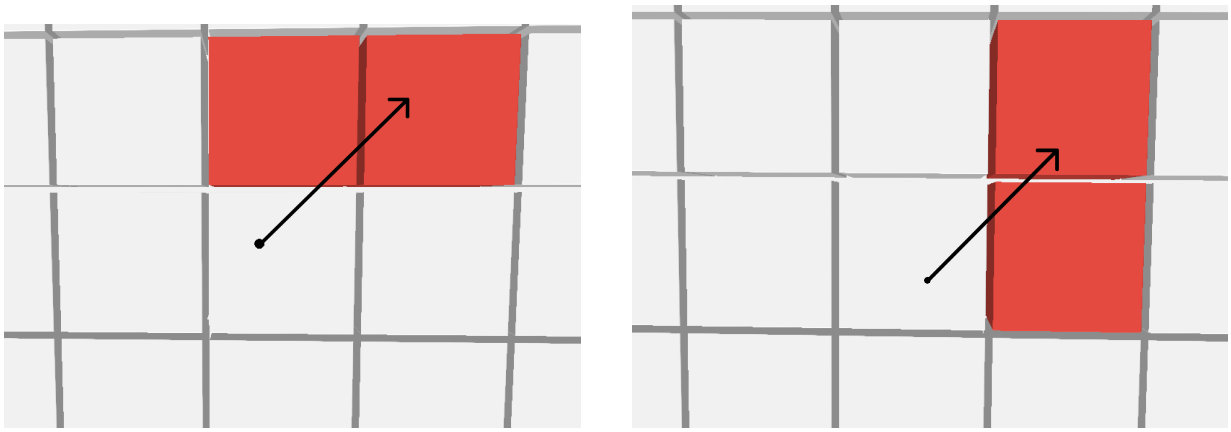
6.3.9 Preskakovanie rohov políčok

Na **Obr. 6.4** je vidieť, že na rohoch vyšších herných políčok vznikajú pri zobrazovaní značné nepresnosti. Dôvodom je, že pri vyhodnocovaní funkcie vzdialenosti sa neráta vždy vzdialenosť od najbližšieho herného políčka, ale od políčka, ktoré je pod pozíciou lúča. V prípade, že je také políčko veľmi nízke, môže byť vzdialenosť od neho dostatočne vysoká na to, aby ďalší krok o takej veľkosti preskočil susediace políčko, ktoré bolo v skutočnosti bližšie k pôvodnej pozícii lúča.

Navrhované riešenie problému je, že okrem vyhodnotenia funkcie vzdialenosti pre políčko pod lúčom sa bude vyhodnocovať aj pre dve najbližšie susediace políčka v smere lúča. To, ktoré dve políčka to budú sa určuje na základe smeru lúča a pozície lúča v rámci políčka pod lúčom.

Cieľom je zistiť z ktorej strany herného políčka lúč vyletí. Susediace políčko na danej strane bude jedným s dvoch dodatočne prehľadávaných políčok. Druhým bude políčko diagonálne od pôvodného políčka. Diagonálne políčko sa vyberá podľa smeru lúča.

Toto riešenie spomalí proces vykresľovania, takže je nutné vymedziť čo najmenšiu množinu prípadov, kedy je ho potrebné použiť. V rámci bakalárskej práce je tento postup aplikovaný len v prípade, že lúč sa nachádza v blízkosti hracej plochy a len ak dosahuje určitú rýchlosť.



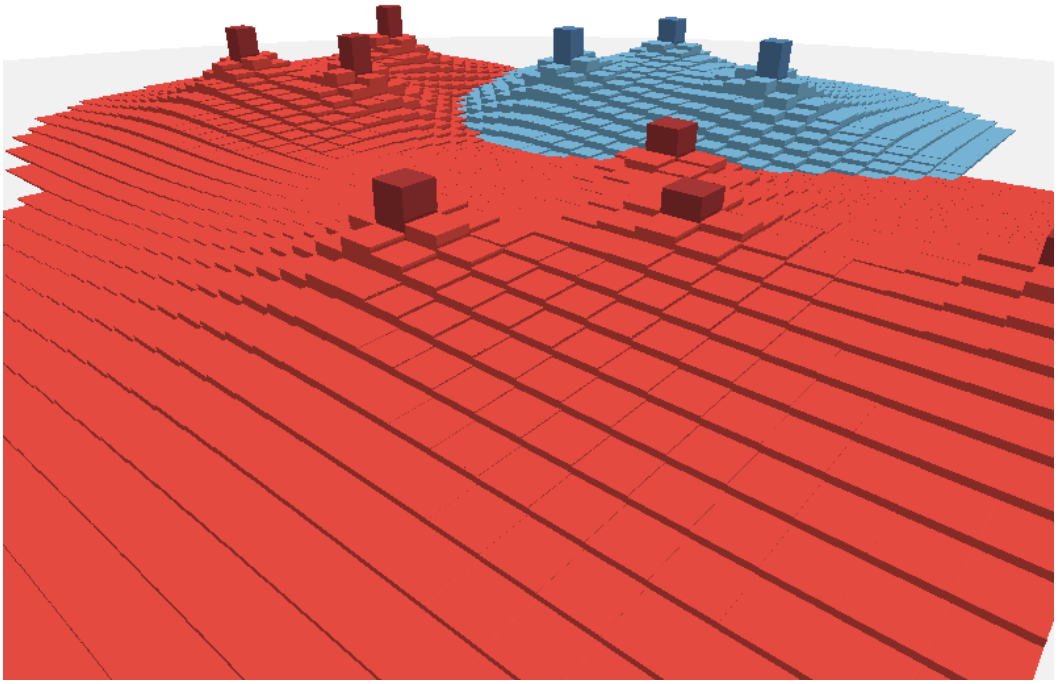
Obr. 6.5: *Výber susediacich políčok na základe smeru a pozície lúča*

```
float xDirection = (r.x > 0 ? 1 : -1);    // smer lúča v x-ovej osi
float zDirection = (r.z > 0 ? 1 : -1);    // smer lúča v z-ovej osi
// Vzdialenosť ku strane v danej osi
float xDistance = xDirection - p.x;
float zDistance = zDirection - p.z;
// Časť vzdialenosti ku strane ktorú prejde lúč za jeden krok
float stepX = r.x / xDistance;
float stepZ = r.z / zDistance;
// Ku strane v x-ovej osi sa dostane ako ku prvej
if (stepX > stepZ)
{
      vec2 neighborOffset = vec2(xDirection, 0);
      vec2 neighbor2Offset = vec2(xDirection, zDirection);
} else      // Ku strane v z-ovej osi sa dostane ako ku prvej
{
      vec2 neighborOffset = vec2(0, zDirection);
      vec2 neighbor2Offset = vec2(xDirection, zDirection);
}

// pozícia suseda ( origin je poloha pôvodného políčka )
vec2 neighborPosition = origin + neighborOffset;
vec2 neighbor2Position = origin + neighbor2Offset;
```

Kód 6.12: *Výpočet polohy susedov v GLSL*

Po implementácii tohto riešenia sú ale stále viditeľné zobrazovacie nepresnosti pri veľkom rozdiele výšok políčok. Doplnok riešenia je spomaľovanie lúčov na polovičnú rýchlosť v prípade, že susediace políčka v smere lúča majú väčšiu výšku a takisto obmedzenie maximálnej rýchlosti nad hraciou plochou. Po tejto úprave je výstup vyhovujúci.



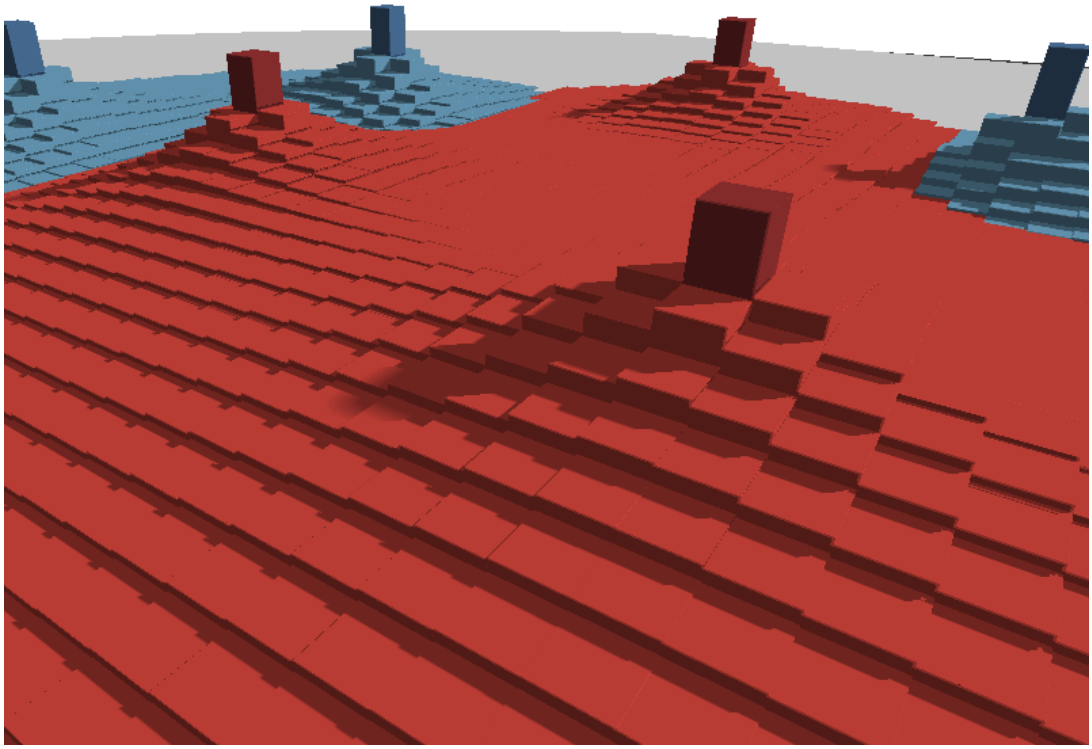
Obr. 6.6: *Zobrazenie hracej plochy bez zobrazovacích nepresností*

6.3.10 Tiene

Jednou z výhod metódy Distance Field Ray Marching je, že ponúka pre každý bod informáciu o celej scéne. To znamená, že je možné preskúmať okolie bodu relatívne jednoducho použitím funkcie vzdialenosti. To sa dá využiť na rýchlu implementáciu tieňov. Stačí, že sa pre skúmaný bod prevedie krokovanie lúča z daného bodu smerom ku svetlu. V prípade, že lúč na niečo narazí, znamená to, že bod je zatienený.

```
float shadow(vec3 ro, vec3 rd, float mint, float maxt )
{
    for( float t=mint; t < maxt; )
    {
        float h = surface(ro + rd*t, rd);
        if( h<0.001 )
            return 0.0;
        t += h;
    }
    return 1.0;
}
```

Kód 6.13: *Funkcia na výpočet tieňovania v GLSL* (prevzaté z [16])



Obr. 6.7: *Implementácia s tieňmi*

6.3.11 Skybox

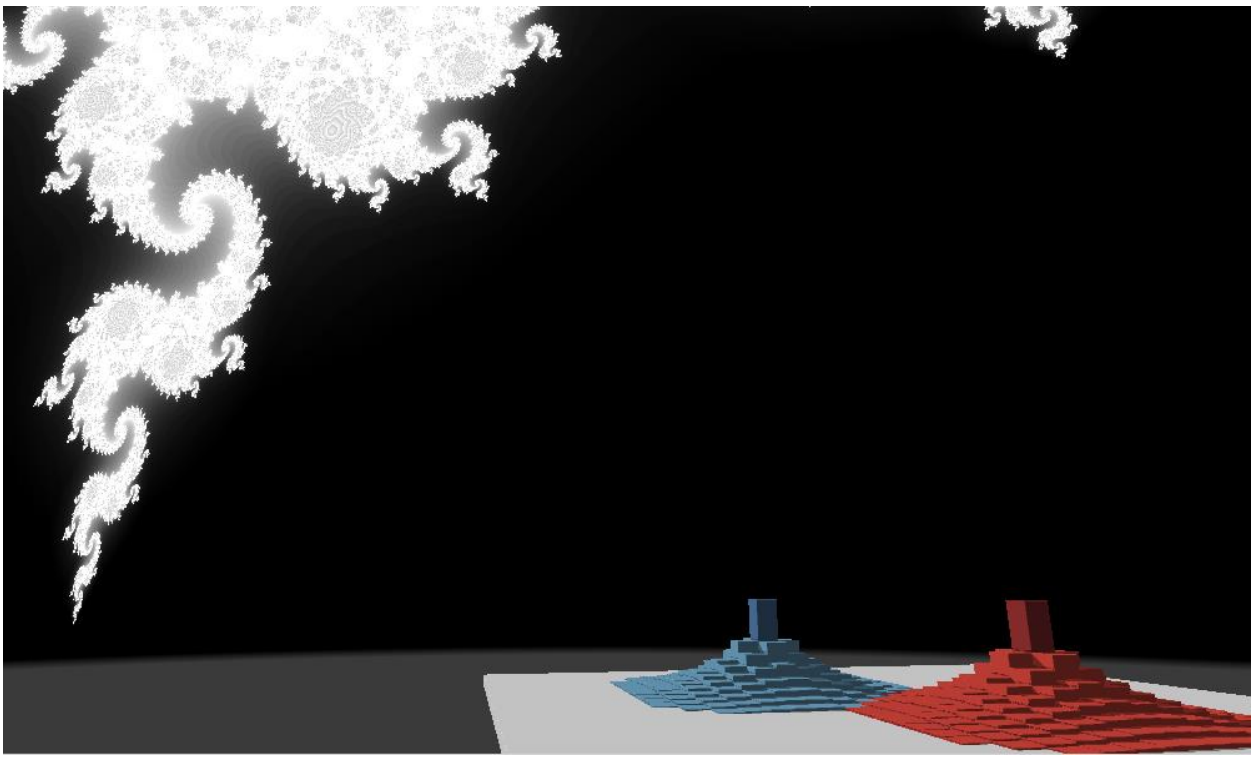
Pri použití zobrazovacej metódy Distance Field Ray Marching je možné jednoducho implementovať procedurálne vytvorený skybox. Stačí v prípade, že lúč presiahne maximálnu vzdialenosť (`maxDist` v **Kóde 6.5**) pridať do scény objekt guľe, ktorá obkolesuje scénu a na ňu zhora premietnuť požadovaný skybox.

V prípade tejto implementácie je skybox čierna „obloha" s premietnutou Juliovou množinou na intervale (-1.6,1.6), ktorá sa mení v čase a rotuje okolo hracej plochy.

```
vec4 SkyBox(vec3 p, vec3 rd)
{
	// posun do stredu hracej plochy
	p.x -= gamesize;
	p.z += gamesize;
	for (int i = 0; i < 20;i++)
	{

		// funkcia vzdialenosti pre guľu, zvnútra
		float d = 190 - length(p);
		if (abs(d) < 0.001)
			break;
		p += rd * d;
	}
	// projekcia bodu z priestoru do roviny do intervalu (-1.6, 1.6)
	vec2 plane = vec2((p.x / 190) * 1.6, (p.z / 190) * 1.6);
	return julia(plane);
}
```

Kód 6.14: *Funkcia generujúca skybox s Juliovou množinou v GLSL*



Obr. 6.8: *Procedurálny skybox – Juliova množina*

6.4 Porovnanie s rasterizáciou

Rasterizácia je v súčasnosti štandardná metóda používaná na vykresľovanie 3D scén. V porovnaní s metódou Distance Field Ray Marching je schopná vykresliť ľubovoľný objekt, pričom v Distance Field Ray Marching je potrebné jeho povrch popísať analyticky. Na druhej strane opakovať objekt do nekonečna objekt v Distance Field Ray Marching nepridáva prakticky žiadne nároky na výpočtový výkon, pričom za použitie rasterizácie požiadavky na výkon stúpajú s každou pridanou inštanciou objektu.

Metódy založené na Ray Castingu sa v súčasnosti nepoužívajú ako štandard, keďže stále majú väčšie požiadavky na výkon ako rasterizácia. Ponúka ale vysokú kvalitu zobrazenia a zaujímavé možnosti, ktoré sa ukazujú na technických demách vyvíjaných nadšencami. Ako je možné vidieť na **Obr. 6.9**, je možné touto metódou dosiahnuť vysokú kvalitu obrazu, ktorá by možno nebola možná za použitia rasterizácie.

Obr. 6.9: *Terén vykreslený metódou Distance Field Ray Marching* (prevzaté z [15])

6.4.1 Porovnanie rýchlosti

Účelom tohto testovania je porovnanie rýchlosti zobrazovania scény metódami rasterizácia a Distance Field Ray Marching. Proces testovania pozostával zo zobrazovania nenulového herného polia zo zamknutou kamerou po dobu 100 snímkov. Na základe toho sa počítala priemerné doba v milisekundách, ktorá bola potrebná na zobrazenie jedného snímku. Vykresľovaný bol pohľad zhora na celú hraciu plochu, viď **Obr. 6.10**. Pre každú zobrazovaciu metódu bolo spustených 6 testov,

s meniacou sa veľkosťou hracej plochy. Testované veľkosti boli 32x32, 64x64, 128x128, 256x256, 512x512 a 1024x1024. Okno v ktorom prebiehal test malo rozlíšenie 1024x600 pixelov.

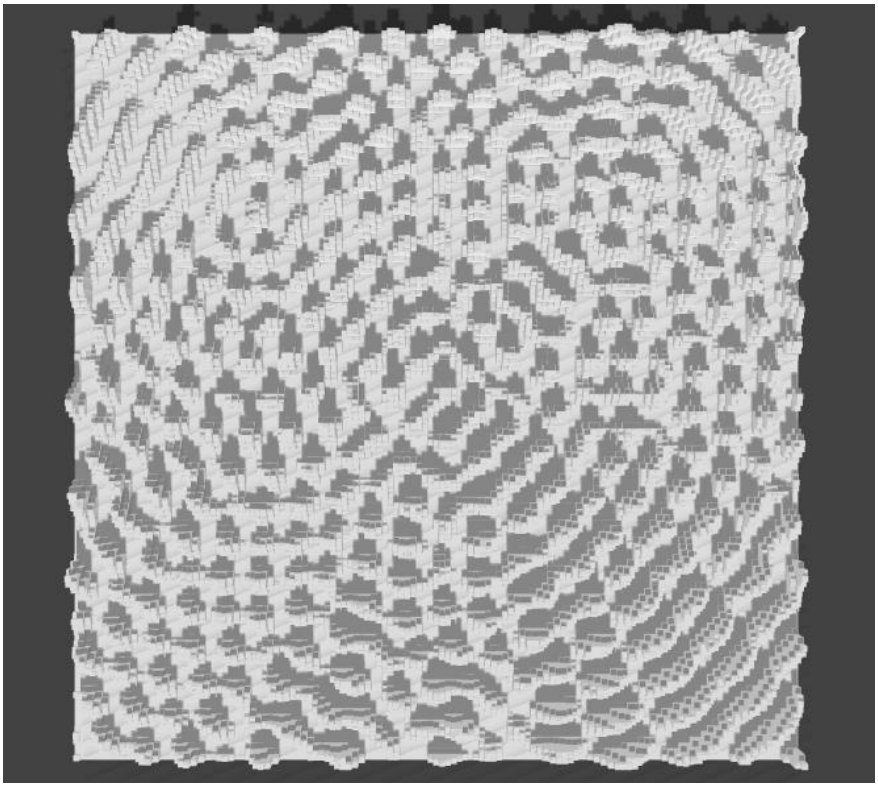
Testy prebiehali na dvoch zostavách, jednej prenosnej – notebook, druhá zostava bola desktop. Špecifikácie zostáv sú nasledovné:

Zostava 1 – notebook:

- **Notebook:** Dell Inspiron 15z
- **Procesor:** Intel Core i7-3537U 2.0GHz
- **Pamäť:** 2x4GB DDR3 1600Mhz
- **Grafická karta:** NVIDIA GeForce GT 630M (verzia ovládačov 334.89)

Zostava 2 – desktop:

- **Procesor:** Intel Core i5-3570 3.4GHz
- **Pamäť:** 8GB, DDR3 1333Mhz
- **Grafická karta:** NVIDIA GeForce GTX 780 (verzia ovládačov 331.82)



Obr. 6.10: *Scéna použitá na testovanie, veľkosť hracej plochy 128x128*

Veľkosť hracieho poľa	32x32	64x64	128x128	256x256	512x512	1024x1024
Rasterizácia	17 ms	22 ms	38 ms	139 ms	623 ms	2923 ms
DFRM	77 ms	76 ms	105 ms	127 ms	151 ms	156 ms

Tab 6.1: Výsledky testovania pre **Zostavu 1**

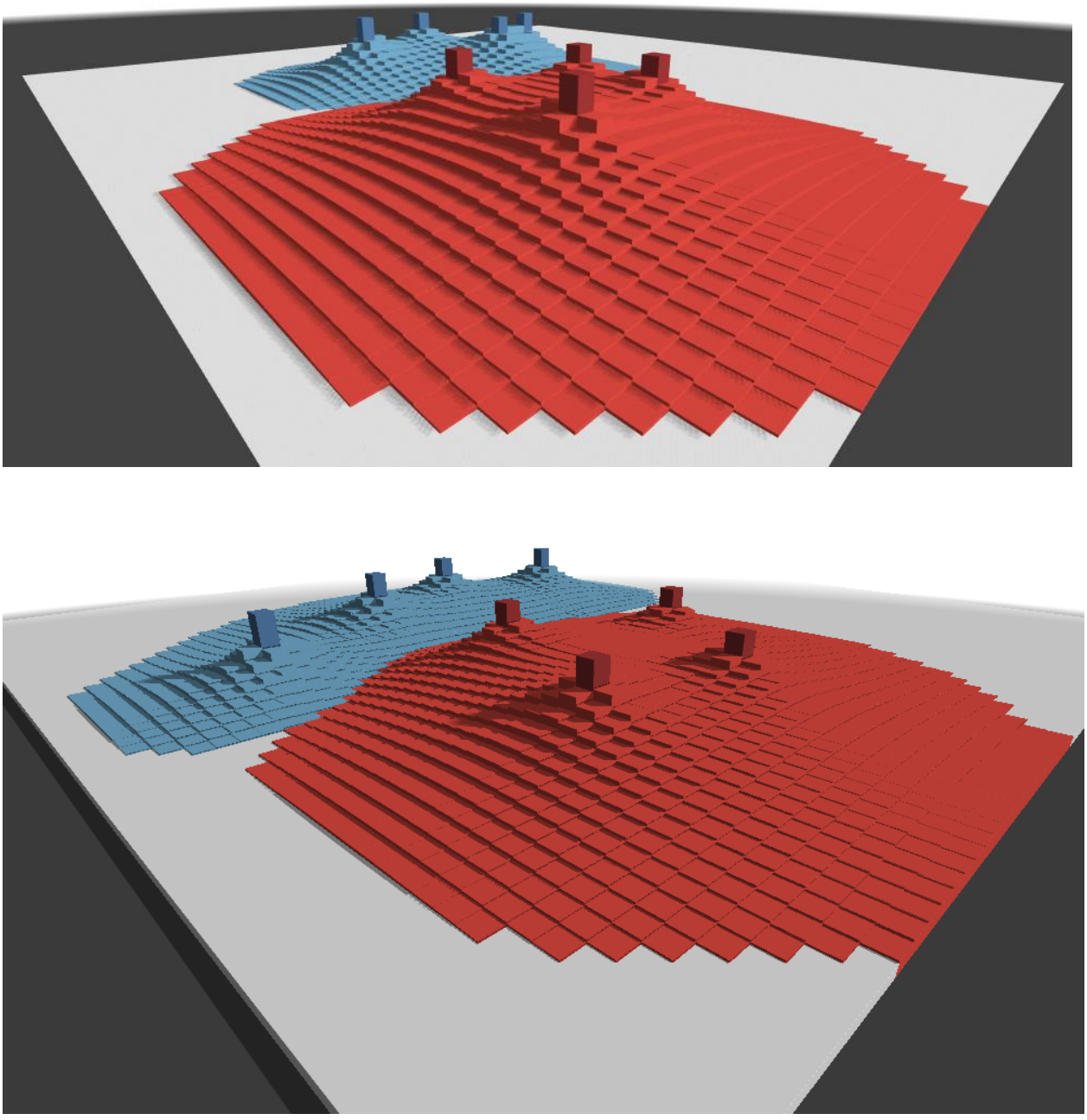
Veľkosť hracieho poľa	32x32	64x64	128x128	256x256	512x512	1024x1024
Rasterizácia	16 ms	16 ms	17 ms	41 ms	157 ms	629 ms
DFRM	16 ms	16 ms	16 ms	16 ms	16 ms	16 ms

Tab 6.2: Výsledky testovania pre **Zostavu 2**

Z výsledkov (**Tab 6.1** a **6.2**) je zrejmé, že pri zobrazovaní veľkého množstva rovnakých objektov je výhodnejšie použiť metódu Distance Field Ray Marching. Na výkonnejšej zostave nemá pri tejto metóde zjavne veľkost herného poľa na rýchlosť zobrazovania veľký vplyv, kedže rýchlosť zobrazovanie je pre všetky veľkosti herného poľa rovnaká. To je pravdepodobne spôsobené tým, že počet lúčov, ktoré sa musia sledovať je nezávislý od veľkosti zobrazovaného herného poľa. Mení sa len vzdialenosť, ktorú musí lúč prekonávať, čo zjavne nemá na výkonnej zostave veľký vplyv. Na zostave č.1 je síce v prípade menšieho herného poľa Distance Field Ray Marching pomalší, no so zvyšovaním veľkostí herného poľa nenarastá čas potrebný na vykreslenie jedného snímku tak rapídne ako je to v prípade rasterizácie. V prípade herného poľa o veľkosti 512x512, alebo 1024x1024 je Distance Field Ray Marching dokonca aj na pomalšej zostave výrazne rýchlejší.

Z výsledkov je možné usúdiť, že na zobrazovanie podobne štrukturovanej scény je v prípade, že to je možné, výhodnejšie použiť Distance Field Ray Marching. Dosahuje podstatne vyššiu rýchlosť zobrazovania a kvalita obrazu je mierne horšia. Implementáciou ambient occlusion a antialiasingu by bolo možné dosiahnuť ešte vyššiu kvalitu obrazu, aj keď by to znamenalo zníženie rýchlosti. Výkonnostná rezerva je totiž v prípade zostavy č.2 dostatočne veľká.

Použitie rasterizácie s cieľom dosiahnuť vyššiu rýchlosť zobrazovania sa oplatí len v prípade menšej hracej plochy (32x32 až 128x128) a to tiež iba na výkonovo slabšej zostave.



Obr. 6.11: *Porovnanie kvality zobrazenia rasterizácie (hore) a DRMF (dole)*

7 Záver

Cieľom bakalárskej práce bolo navrhnúť a implementovať v OpenGL jednoduchú strategickú hru, ktorej herná logika aj zobrazovanie bude bežať na grafickej karte. K tomu bolo potrebné sa s rozhraním OpenGL oboznámiť, získať prehľad o tvorbe procedurálnych textúr a preštudovať vhodné zobrazovacie metódy.

Myslím, že tieto všetky body boli dosiahnuté. Preštudované metódy pre tvorbu procedurálnych textúr boli napríklad Perlinov šum, Voroného diagramy, fraktály, alebo celulárne automaty. Zo zobrazovacích metód boli naštudované metódy založené na metóde Ray Casting a rasterizácia. Na základe toho bol navrhnutý a implementovaný celulárny automat realizujúci hernú logiku hry a takisto fragment shader, ktorý realizoval zobrazovanie metódou Distance Field Ray Marching. Spracovanie vstupu hry prebieha na procesore, keďže grafická karta nemá podobné možnosti.

V rámci implementácie shaderu pre zobrazovanie bolo potrebné riešiť problém rôznej výšky herných políčok a následne optimalizovať tento shader, aby dosahoval požadovanú rýchlosť a kvalitu. Najproblematickejším sa ukázalo byť zobrazovanie veľkých rozdielov vo výške herných políčok, ktorého riešenie pridalo aj po optimalizácii výrazné požiadavky na výkon grafickej karty. Na druhej strane implementácia tieňov a skyboxu vyžadovala minimálnu pozornosť. Takisto bolo pre porovnanie implementované zobrazenie pomocou rasterizácie, ktoré sa ukázalo byť výrazne pomalšie, väčšiu rýchlosť dosahovalo len v prípade menšieho herného poľa a slabej grafickej karty.

Z pohľadu ďalšieho vývoja je možné ďalej optimalizovať zobrazovací proces a implementovať komplexnejšie osvetlenie s ambient occlusion. Takisto je možné zvýšiť výslednú kvalitu obrazu antialiasingom a post-processing efektami. Možné rozšírenie je procedurálne generovanie terénu v okolí hracej plochy a komplexnejší skybox, pomocou Perlin noise alebo Celulárnych automatov. Zaujímavé by bolo takisto implementovať rozhranie, ktorým môže shader hernej logiky komunikovať s procesorom a teda kontrola vstupu a vyhodnocovanie stavu hry by mohlo prebiehať taktiež na shadery.

8 Literatúra

[1] WRIGHT, Richard S Jr., HAEMEL, Nicholas, SELLERS, Graham, LIPCHAK Benjamin. *OpenGL SuperBible, Fifth Edition, Comprehensive Tutorial and Reference* [online]. ADDISON-WESLEY, 2011. Dostupné na: http://ldc.usb.ve/~vtheok/cursos/ci4321/pdfs/02.pdf

[2] JÖNSSON, Andreas, Generating Perlin Noise [online]. Dostupné na: http://www.angelcode.com/dev/perlin/perlin.html

[3] ELIAS, Hugo. Perlin Noise [online]. Dostupné na: http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

[4] ZUCKER, Matt. The Perlin Noise FAQ [online]. Dostupné na: http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

[5] QUÍLEZ, Iñigo. Domain Warping [online]. Dostupné na: http://www.iquilezles.org/www/articles/warp/warp.htm

[6] WEISSTEIN, Eric W. Voronoi Diagram. From MathWorld--A Wolfram Web Resource. Dostupné na: http://mathworld.wolfram.com/VoronoiDiagram.html

[7] QUÍLEZ, Iñigo. Cellular Effect - Voronoi [online]. Dostupné na: http://www.iquilezles.org/www/articles/cellularffx/cellularffx.htm

[8] Mandelbrot Set [online]. Wikipedia. Dostupné na: http://en.wikipedia.org/wiki/Mandelbrot_set

[9] Julia Set [online]. Wikipedia. Dostupné na: http://en.wikipedia.org/wiki/Julia_set

[10] Cellular Automaton [online]. Wikipedia. Dostupné na: http://en.wikipedia.org/wiki/Cellular_automaton

[11] WALTER, Marcelo, FOURNIER, Alain, REIMERS Mark. *Clonal Mosaic Model for the Synthesis of Mammalian Coat Patterns* [online]. Dostupné na: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.6.1013&rep=rep1&type=pdf

[12] BUCK, Jamis. Ray Tracing Algorithm [online]. Dostupné na: http://reocities.com/SiliconValley/haven/5114/raytracing.html

[13] QUÍLEZ, Iñigo. Terrain Raymarching [online]. Dostupné na: http://www.iquilezles.org/www/articles/terrainmarching/terrainmarching.htm

[14] QUÍLEZ, Iñigo. Rendering Worlds with Two Triangles with raytracing on the GPU in 4096 bytes [online].
Dostupné na: http://www.iquilezles.org/www/material/nvscene2008/rwwtt.pdf

[15] QUÍLEZ, Iñigo. [online].
Dostupné na: http://www.iquilezles.org/blog/?p=2422

[16] QUÍLEZ, Iñigo. Free penumbra shadows for raymarching distance fields [online].
Dostupné na: http://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm

Príloha A

Plagát